

Computer Aritmetiek

Pascal Vyncke

OPGAVE

- (1) Om 7 beduidende cijfers voor $\ln 2$ uit te rekenen door gebruik te maken van de reeksontwikkeling van $\ln(1+x)$ benodigt men een ongelooijk groot aantal termen. Hoeveel termen heb je nodig als je $\ln 2$ wil programmeren door gebruik te maken van de reeksontwikkeling van $\ln \frac{1+x}{1-x}$ (het is voor het project voldoende om alleen de truncatiefout in rekening te brengen)?
- (a) Is deze x exact voorstelbaar?
- (b) Programmeer de berekening van $\ln 2$ zo stabiel mogelijk in enkele en dubbele precisie.
- (c) Heb je in enkele precisie effectief 7 beduidende cijfers of wordt de afrondingsfout te groot?
- (2) Voor $x_0 > -1$ geldt dat de rij x_n met $x_{n+1} = \frac{1}{2} \left(x_n + \frac{1}{x_n} \right)$ convergeert naar $\ln(1+x_0)$.
- (a) Programmeer deze recursie en vergelijk met $\ln(1+x_0)$.
- (b) Herschrijf de recursie zodat de berekeningen betrouwbaarder worden.
- (3) We willen $f(x) = (x - \sin x)/x^3$ berekenen voor $x = 10^{-k}$ met $k = 0; \dots; 15$.
- (a) Welke waarden bekom je als je $f(x)$ programmeert zoals opgegeven?
- (b) Waaraan is $\lim_{x \rightarrow 0} f(x)$ gelijk?
- (c) Hoe moet je de berekening van $f(x)$ voor x klein organiseren?

Oefening 1

1. a)

Om de uitkomst van $\ln 2$ te verkrijgen via de reeksontwikkeling van $\ln \left[\frac{(1+x)}{(1-x)} \right]$ is x niet exact voorstelbaar (niet in binaire representatie, niet in decimale representatie).

1. b)

De berekening van $\ln 2$ via de reeksontwikkeling van $\ln \left[\frac{(1+x)}{(1-x)} \right]$ is het volgende.

Volgens de definitie van $\ln$ is het volgende waar:

$$\ln \left[\frac{(1+x)}{(1-x)} \right] = \ln(1+x) - \ln(1-x)$$

We kunnen $\ln(1+x)$ en $\ln(1-x)$ omzetten in elk hun eigen reeksontwikkeling, en vervolgens van elkaar aftrekken en zo een totale reeksontwikkeling verkrijgen die $\ln \left[\frac{(1+x)}{(1-x)} \right]$ voorstelt.

Hiermee komen we echter niet verder, aangezien we dan twee keer de $\ln$ moeten berekenen en aftrekken (en aftrekking is “gevaarlijk” voor computers op basis van nauwkeurigheid; om aan de opgave te voldoen, stabiel programmeren dienen we dus de aftrekking te vermijden). Bovendien wordt in de opgave ook aangegeven dat de reeksontwikkeling van $\ln(1+x)$ al zeer groot is en dat we het hiermee niet gaan halen.

Dus ik zoek het via een andere weg de reeksontwikkeling van $\ln \left[\frac{(1+x)}{(1-x)} \right]$.

De reeksontwikkeling van $\ln [(1+x)/(1-x)]$ is:

We geven er een functie $f(x)$ als naam aan

$$f(x) = \ln [(1+x)/(1-x)]$$

via afleiden van $f(x)$ krijgen we:

$$f'(x) = 2 / (1-x^2)$$

de reeksontwikkeling hiervan is:

$$2 + 2x^2 + 2x^4 + 2x^6 + 2x^8 + 2x^{10} + \dots$$

door terug te integreren term per term van bovenstaande reeksontwikkeling, krijgen we:

$$2x + (2x^3)/3 + (2x^5)/5 + (2x^7)/7 + \dots$$

$$= 2(x + x^3/3 + x^5/5 + x^7/7 + \dots) \text{ en dit is de reeksontwikkeling van } \ln [(1+x)/(1-x)]$$

We dienen dus $2(x + x^3/3 + x^5/5 + x^7/7 + \dots)$ te programmeren en berekenen.

Deze formule is echter geldig voor eender welke $\ln$. De opgave vraagt enkel de $\ln 2$.

Omdat de formule enkel geldig is voor een willekeurige x tussen -1 en 1, gaan we dit omzetten. Het is namelijk mogelijk om elk getal hierin om te zetten.

Het getal 2 ligt niet tussen de grenzen van -1 en 1. Echter, door dit te inverteren, dus 2^{-1} , komt dit wél te liggen in -1 en 1, namelijk $1/2$.

We kunnen dit gebruiken omwille van volgende regel van logaritmen:

$$-\ln(x) = \ln(1/x)$$

Zodus, kunnen we de $\ln(2)$ gaan voorstellen, door gebruik te maken van $1/2$ en dan onze uitkomst uiteindelijk het teken om te draaien (dus positief wordt negatief, en omgekeerd).

Voor gebruik te maken van onze formule, bekomen we dus dat $x = -1/3$

Dit getal is (zoals in onderdeel a van deze oefening gezegd), niet exact voorstelbaar.

(Namelijk in decimale getallen is dit -0,3333333333333333... en ook in binaire voorstelling, namelijk -.0101010101010101...)

We kunnen dus het volgende stellen:

$$\ln 2 = \ln [(1+(-1/3))/(1-(-1/3))] = 2((-1/3) + (-1/3)^3/3 + (-1/3)^5/5 + (-1/3)^7/7 + \dots)$$

Een narekening van dit resultaat in een wiskundig programma (Mathlab), geeft aan dat

$$2((-1/3) + (-1/3)^3/3 + (-1/3)^5/5 + (-1/3)^7/7) \approx -0,693134757$$

De vraag wat $\ln 2$ nu eigenlijk is, krijgen we als antwoord van het programma

0,6931471805599450, waarbij de cijfertjes in het vet dezelfde zijn als met ons vorige resultaat verkregen, met andere woorden zitten we daarmee al aan 5 juiste cijfers.

Hiermee is de oplossing gecontroleerd, en kunnen we nu deze oplossing gaan implementeren. Ik heb gekozen om dit in C++ te doen.

Het *stabiel* programmeren van de $\ln 2$ is reeds gedeeltelijk opgelost. Ik gebruik nu al de reeksontwikkeling en er worden enkel maar optellingen gebruikt, en geen aftrekkingen (wat bij de allereerste oplossing, namelijk $\ln [(1+x)/(1-x)] = \ln (1+x) - \ln (1-x)$ niet zo was).

Een tweede regel dat ik hanteer is dat we éérst de kleinste getallen moeten optellen, en dan pas de grote. Dus de reeksontwikkeling in feite omdraaien (wat met optellen volgens de wiskunde perfect mag). Dus:

$$\ln [(1+(-1/3))/(1-(-1/3))] = 2(\dots + (-1/3)^7/7 + (-1/3)^5/5 + (-1/3)^3/3 + (-1/3))$$

Dit blijft dus hetzelfde als resultaat hebben wiskundig gezien, met de computer uitgerekend verhoogt dit echter de nauwkeurigheid (beter gezegd: laat de (kans op) afrondingsfouten dalen).

We willen 7 beduidende cijfers hebben, we nemen daarom 8 termen.

De C++ code:

Voor single precision:

```
#ifndef INC_IOSTREAM
#include <iostream>
#define INC_IOSTREAM
#endif

#ifndef INC_STRING
#define INC_STRING
#include <string>
#endif

#ifndef INC_MATH
#define INC_MATH
#include <math.h>
#endif

using namespace std;

int main () {
    float x=0;
    float oplossing=0;
    float tussenoplossing=0;
    int macht;

    macht = 13; // Komt overeen met 6 termen ((6*2) + 1)
    x = 1;
    x = -(x / 3); // x = -0.333333333333

    while(macht > 0) {
        // Een tussenoplossing berekenen
```

```

        tussenoplossing = (pow(x,macht) / (macht));

        // Optellen met onze vorige oplossing
        oplossing = oplossing + tussenoplossing;

        macht = macht - 2;
    }

    oplossing = (oplossing * 2); // Oplossing vermenigvuldigen met 2
    oplossing = -oplossing; // Positief wordt negatief, en omgekeerd

    // Nu onze gevonden oplossing outputten
    cout.precision(50);
    cout << oplossing;
}

```

Dit geeft uitgevoerd op een Pentium 4 met het Microsoft Windows platform het volgende als resultaat: 0.69314718246459961

Hierbij zijn er 7 beduidende cijfers:

0.69314718246459961

Voor double precision

```

#ifndef INC_Iostream
#include <iostream>
#define INC_Iostream
#endif

#ifndef INC_STRING
#define INC_STRING
#include <string>
#endif

#ifndef INC_MATH
#define INC_MATH
#include <math.h>
#endif

using namespace std;

int main () {
    double x=0;
    double oplossing=0;
    double tussenoplossing=0;
    int macht;

    macht = 15; // Komt overeen met 7 termen ((7*2) + 1)
    x = 1;
    x = -(x / 3); // x = -0.333333333333

    while(macht > 0) {
        // Een tussenoplossing berekenen
        tussenoplossing = (pow(x,macht) / (macht));

        // Optellen met onze vorige oplossing
    }
}

```

```

        oplossing = oplossing + tussenoplossing;

        macht = macht - 2;
    }

    oplossing = (oplossing * 2); // Oplossing vermenigvuldigen met 2
    oplossing = -oplossing; // Positief wordt negatief, en omgekeerd

    // Nu onze gevonden oplossing outputten
    cout.precision(50);
    cout << oplossing;
}

```

Dit geeft uitgevoerd op een Pentium 4 met het Microsoft Windows platform het volgende als resultaat: 0.693147179548241230

Hierbij zijn er 7 beduidende cijfers:

0.693147179548241230

1.c)

In single precision hebben we net voldoende precisie om 7 beduidende cijfers te hebben. We kunnen dit uitrekenen met de formule. Namelijk:

$$7 \text{ beduidende cijfers} \Leftrightarrow \left| \frac{x - \tilde{x}}{x} \right| \leq \frac{1}{2} \cdot 10^{-7+1}$$

Dit uitgerekend brengt dat:

$$7 \text{ beduidende cijfers} \Leftrightarrow 0,00000000274797869995 \leq 0,0000005$$

En dit klopt want dit is inderdaad kleiner.

Dit valt ook te vergelijken:

Single precision: **0.6931471**82464699610

Double precision: **0.6931471**79548241230

Juist ná wat vetjes gedrukt is, zijn dit andere cijfers. De cijfers van double precision zijn echter wel correct, die van single niet meer.

Samengevat: het lukt juist om 7 beduidende cijfers te hebben met single precision.

Oefening 2

2a)

De recursie heb ik geprogrammeerd in C++.

De C++ programmacode:

```
#ifndef INC_IOSTREAM
#include <iostream>
#define INC_IOSTREAM
#endif

#ifndef INC_STRING
#define INC_STRING
#include <string>
#endif

#ifndef INC_MATH
#define INC_MATH
#include <math.h>
#endif

using namespace std;

double Oefening2(int n, double x0) {
    if (n > 0) {
        return ( pow(2,n) * (sqrt(1 + ((1 / pow(2,n-1)) *
Oefening2((n-1), x0)))-1));
    }
    else
    {
        return x0;
    }
}

int main () {
    double x0;

    // Bereken ln(1+2)
    x0 = 2;
    cout.precision(50);
    cout << "Uitkomst voor x0=" << x0 << ":" << Oefening2(10,x0) << endl;
    cout << "Uitkomst voor x0=" << x0 << ":" << Oefening2(15,x0) << endl;
    cout << "Uitkomst voor x0=" << x0 << ":" << Oefening2(20,x0) << endl;
    cout << "Uitkomst voor x0=" << x0 << ":" << Oefening2(25,x0) << endl;
    cout << "Uitkomst voor x0=" << x0 << ":" << Oefening2(30,x0) << endl;
    cout << endl << endl;
    cout << "Vergelijking met log(1+x0)" << log(1+x0) << endl;

    // Bereken ln(1+3)
    x0 = 3;
    cout.precision(50);
    cout << "Uitkomst voor x0=" << x0 << ":" << Oefening2(10,x0) << endl;
    cout << "Uitkomst voor x0=" << x0 << ":" << Oefening2(15,x0) << endl;
    cout << "Uitkomst voor x0=" << x0 << ":" << Oefening2(20,x0) << endl;
    cout << "Uitkomst voor x0=" << x0 << ":" << Oefening2(25,x0) << endl;
    cout << "Uitkomst voor x0=" << x0 << ":" << Oefening2(30,x0) << endl;
    cout << endl << endl;
    cout << "Vergelijking met log(1+x0)" << log(1+x0) << endl;
}
```

Bovenstaande programma gaat de $\ln(1+2)$ en van $\ln(1+3)$ uitrekenen.

Bovenstaande programma geeft op een Pentium 4 met het Microsoft Windows platform het volgende als resultaat:

Uitkomst voor $x_0=2$: 1.0992018300289601000000000000
Uitkomst voor $x_0=2$: 1.0986307054554345000000000000
Uitkomst voor $x_0=2$: 1.0986128642689437000000000000
Uitkomst voor $x_0=2$: 1.0986123085021973000000000000
Uitkomst voor $x_0=2$: 1.0986120700836182000000000000
Vergelijking met $\log(1+x_0) = 1.0986122886681098000000000000$

Uitkomst voor $x_0=3$: 1.3872331695188223000000000000
Uitkomst voor $x_0=3$: 1.3863236860561301000000000000
Uitkomst voor $x_0=3$: 1.3862952776253223000000000000
Uitkomst voor $x_0=3$: 1.3862943947315216000000000000
Uitkomst voor $x_0=3$: 1.3862943649291992000000000000
Vergelijking met $\log(1+x_0) = 1.3862943611198906000000000000$

Het eerste wat we opmerken is dat naarmate we meer recursies doen, het resultaat nauwkeuriger en nauwkeuriger wordt. In mijn programma ga ik tot 30 stappen. Hiermee kan ik, in vergelijking met de “echte” uitkomst van het logaritme, tot 8 cijfers nauwkeurig zijn:

Uitkomst voor $x_0=3$ na 30 recursies: **1.38629436**49291992
Vergelijking met $\log(1+x_0) =$ **1.38629436**11198906

2b)

In oefening a is de berekening met de recursie puur ingegeven zoals in de opgave, zonder rekening te houden met bepaalde optimalisaties, regels of om de nauwkeurigheid te verhogen (en afrondingsfouten te vermijden).

Dit wordt nu wel toegepast, om zo een mogelijk beter resultaat te verkrijgen.

Origineel volgens opgave:

$$x_{n+1} = 2^{n+1} \left(\sqrt{1 + 2^{-n} x_n} - 1 \right)$$

Om dit effectief te programmeren, werd dit omgezet naar het volgende:

$$x_n = 2^n \left(\sqrt{1 + 2^{-(n-1)} x_{n-1}} - 1 \right)$$

Of met andere woorden werd $n+1$ gelijk aan n . We roepen namelijk in ons programma x_n op en niet x_{n+1} .

Met de eigenschappen van de wortels, kunnen we dit omvormen:

$$x_n = 2^n \left(\sqrt{1 + 2^{-(n-1)} x_{n-1}} - \sqrt{1} \right)$$

En dan gebruiken we een eigenschap van de wortels om dit te herschrijven:

$$x_n = 2^n \left(\frac{(1 + 2^{-(n-1)} x_{n-1}) - 1}{\sqrt{1 + 2^{-(n-1)} x_{n-1}} + 1} \right)$$

We gaan nu de 1 – 1 schrappen in de teller:

$$x_n = 2^n \left(\frac{2^{-(n-1)} x_{n-1}}{\sqrt{1 + 2^{-(n-1)} x_{n-1}} + 1} \right)$$

Nu kunnen we de macht $2^{-(n-1)}$ uit onze vergelijking halen:

$$x_n = 2^n \cdot 2^{-(n-1)} \left(\frac{x_{n-1}}{\sqrt{1 + 2^{-(n-1)} x_{n-1}} + 1} \right)$$

Of met andere woorden:

$$x_n = \frac{2^n}{2^{(n-1)}} \cdot \left(\frac{x_{n-1}}{\sqrt{1 + 2^{-(n-1)} x_{n-1}} + 1} \right)$$

En dit is gelijk aan:

$$x_n = 2 \cdot \left(\frac{x_{n-1}}{\sqrt{1 + 2^{-(n-1)} x_{n-1}} + 1} \right)$$

Hierdoor hebben we deze grote machten en delingen weggewerkt, waardoor er een hogere nauwkeurigheid wordt verkregen. Dit bovenstaande kunnen we nu gaan programmeren.

In C++ wordt dit:

```
#ifndef INC_IOSTREAM
#include <iostream>
#define INC_IOSTREAM
#endif

#ifndef INC_STRING
#define INC_STRING
#include <string>
#endif

#ifndef INC_MATH
#define INC_MATH
#include <math.h>
#endif

using namespace std;

double Oefening2b(int n, double x0) {
    double temp0, temp1, temp2, temp3, temp4, temp5;
    if (n > 0) {
        return (((Oefening2b((n-1), x0)))/(sqrt(1 + ((1 / pow(2,n-1)) *
Oefening2b((n-1), x0))) + 1))*2);
    }
    else
    {
        return x0;
    }
}

int main () {
    double x0;

    // Bereken ln(1+2)
    x0 = 2;
    cout.precision(50);
    cout << "Uitkomst voor x0=" << x0 << ":" << Oefening2b(10,x0) << endl;
    cout << "Uitkomst voor x0=" << x0 << ":" << Oefening2b(15,x0) << endl;
    cout << "Uitkomst voor x0=" << x0 << ":" << Oefening2b(20,x0) << endl;
    cout << "Uitkomst voor x0=" << x0 << ":" << Oefening2b(25,x0) << endl;
    cout << "Uitkomst voor x0=" << x0 << ":" << Oefening2b(30,x0) << endl;
    cout << endl << endl;
    cout << "Vergelijking met log(1+x0)" << log(1+x0) << endl;

    // Bereken ln(1+3)
    x0 = 3;
    cout.precision(50);
    cout << "Uitkomst voor x0=" << x0 << ":" << Oefening2b(10,x0) << endl;
    cout << "Uitkomst voor x0=" << x0 << ":" << Oefening2b(15,x0) << endl;
    cout << "Uitkomst voor x0=" << x0 << ":" << Oefening2b(20,x0) << endl;
    cout << "Uitkomst voor x0=" << x0 << ":" << Oefening2b(25,x0) << endl;
    cout << "Uitkomst voor x0=" << x0 << ":" << Oefening2b(30,x0) << endl;
    cout << endl << endl;
    cout << "Vergelijking met log(1+x0)" << log(1+x0) << endl;
}
```

De bekomen recursie, dus:

$$x_n = 2 \cdot \left(\frac{x_{n-1}}{\sqrt{1 + 2^{-(n-1)} x_{n-1}} + 1} \right)$$

Kan terug herschreven worden in de vorm zoals in de opgave:

$$x_{n+1} = 2 \cdot \left(\frac{x_n}{\sqrt{1 + 2^{-n} x_n} + 1} \right)$$

Deze geeft betere waarden terug.

Oefening 3

3a)

De te programmeren functie is:

$$f(x) = \frac{x - \sin x}{x^3}$$

Ook deze oefening maakte ik in C++. De programmacode is als volgt:

```
#ifndef INC_Iostream
#include <iostream>
#define INC_Iostream
#endif

#ifndef INC_STRING
#define INC_STRING
#include <string>
#endif

#ifndef INC_MATH
#define INC_MATH
#include <math.h>
#endif

using namespace std;

double Oefening3a(double x) {
    return ((x - sin(x))/pow(x, 3));
}

int main () {

    int k = 0;
    double x;

    cout.precision(50);

    while (k <= 15) {
        x = (1 / pow(10, k));
        cout << "f(x) met x=10^(-" << k << ") : " << Oefening3a(x) << endl;
        k++;
    }
}
```

Dit geeft als output:

```
f(x) met x=10^(-0) : 0.15852901519210350000000000000000
f(x) met x=10^(-1) : 0.16658335317184766000000000000000
f(x) met x=10^(-2) : 0.16665833335317090000000000000000
f(x) met x=10^(-3) : 0.16666658333286700000000000000000
f(x) met x=10^(-4) : 0.16666666585240510000000000000000
f(x) met x=10^(-5) : 0.16666666955220400000000000000000
f(x) met x=10^(-6) : 0.16666657069354740000000000000000
f(x) met x=10^(-7) : 0.16666396873236397000000000000000
f(x) met x=10^(-8) : 0.16640547479094109000000000000000
```

$f(x)$ met $x=10^{(-9)}$: 0.20194839173657897000000000000000
 $f(x)$ met $x=10^{(-10)}$: 0.14154618974164529000000000000000
 $f(x)$ met $x=10^{(-11)}$: 0.18144524316816599000000000000000
 $f(x)$ met $x=10^{(-12)}$: 0.15543699938875519000000000000000
 $f(x)$ met $x=10^{(-13)}$: 0.11510167293962097000000000000000
 $f(x)$ met $x=10^{(-14)}$: 0.16639556372238837000000000000000
 $f(x)$ met $x=10^{(-15)}$: 0.18099898212263632000000000000000

3b)

De $\lim_{x \rightarrow 0} f(x)$ berekenen we als volgt.

$$\lim_{x \rightarrow 0} f(x) = \lim_{x \rightarrow 0} \left(\frac{x - \sin x}{x^3} \right)$$

Hiervan de limiet nemen is echter niet mogelijk, we hebben een regel/eigenschap nodig om dit te verkrijgen. De regel hiervoor nodig is de regel van L'Hopital:

$$\lim_{x \rightarrow 0} \left(\frac{x - \sin x}{x^3} \right) = \lim_{x \rightarrow 0} \left(\frac{(x - \sin x)'}{(x^3)'} \right)$$

We gaan dit dus afleiden. We verkrijgen:

$$\lim_{x \rightarrow 0} \left(\frac{1 - \cos x}{3x^2} \right)$$

We passen opnieuw de regel van L'Hopital toe:

$$\lim_{x \rightarrow 0} \left(\frac{(1 - \cos x)'}{(3x^2)'} \right) = \lim_{x \rightarrow 0} \left(\frac{\sin x}{6x} \right)$$

We passen nog een laatste keer de regel van L'Hopital toe:

$$\lim_{x \rightarrow 0} \left(\frac{(\sin x)'}{(6x)'} \right) = \lim_{x \rightarrow 0} \left(\frac{\cos x}{6} \right)$$

Hiervan de limiet berekend is gelijk aan:

$$\lim_{x \rightarrow 0} f(x) = \lim_{x \rightarrow 0} \left(\frac{\cos x}{6} \right) = \frac{1}{6} \cong 0,16666666666666666666666666666667$$

3c)

De functie is als volgt:

$$f(x) = \frac{x - \sin x}{x^3}$$

Waarbij x heel erg klein wordt, namelijk machten van 10^{-k} met $k = 1, 2, \dots, 15$.

Om te laten zien hoe x verloopt:

1
0,1
0,01
0,001
0,0001
0,00001
0,000001
0,0000001
0,00000001
0,000000001
0,0000000001
0,00000000001
0,000000000001
0,0000000000001
0,00000000000001
0,000000000000001
0,0000000000000001

We dienen onze berekening zodanig te organiseren dat we rekening houden met een aantal basisregels. Zo zien we dat x^3 minder klein is dan $\sin x$ voor hele kleine waarden van x. Dit kunnen we gebruiken. Bovendien is getallen aftrekken steeds zeer gevaarlijk, in het bijzonder indien het een relatief groot getal aftrekken is van een relatief klein getal. Ook een relatief groot getal delen door een relatief klein getal is niet goed en beter te vermijden. Bij het aftrekken moeten we rekening houden dat het kleinste getal afgetrokken wordt van het grootste getal indien mogelijk.

Het efficiëntste om het uit te rekenen is dus (in C++ code):

```
double Oefening3(double x) {  
    return ((x + sin(-x))/pow(x, 3));  
}
```

Minder efficiënt zijn dus:

```
double Oefening3b(double x) {  
    return (1/pow(x, 3)) * (x - sin(x));  
}  
  
double Oefening3b(double x) {  
    return ((1/pow(x, 2)) - (sin(x)/pow(x, 3)));  
}  
  
double Oefening3b(double x) {  
    return (1 - (sin(x)/x)) * (1/pow(x, 2));  
}
```

```

double Oefening3b(double x) {
    return (1/pow(x,3))*(sin(-x)+x);
}

double Oefening3b(double x) {
    return ((sin(-x)/pow(x,3)) + (1/pow(x,2)));
}

double Oefening3b(double x) {
    return ((sin(-x) + x)/pow(x, 3));
}

double Oefening3b(double x) {
    return ((sqrt((1-cos(2x))/2) + x)/pow(x, 3));
}

```

Om tot de juiste oplossing te komen dienen we de sinus om te vormen naar de reeksontwikkeling van de sinus. Hierdoor krijgen we betere resultaten uiteindelijk en krijgen we minder fouten waardoor ons resultaat wél goed wordt.

De reeksontwikkeling van de sinus is als volgt:

$$\sin x = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!} \text{ voor alle } x$$

We dienen hiervoor in C++ ook nog de faculteit te programmeren, dit is eenvoudig te doen met recursie (maar kan ook indien nodig zonder recursie gemaakt worden en dan met een lus).

Dit in C++ omgezet geeft:

```
double faculteit (int i) {
    if (i <= 0) {
        return 1;
    }
    else
    {
        return i * faculteit(i-1);
    }
}

double SinReeks(double x) {
    int i;
    double oplossing;
    double tussenoplossing;
    double teken;

    teken = 1;
    i = 1;
    oplossing = 0;
    while (i < 9) {
        tussenoplossing = pow(x, i);
        tussenoplossing = (tussenoplossing / faculteit(i));
        if (teken == 2) {
            teken = 1;
            tussenoplossing = - tussenoplossing;
        }
        else
        {
            teken = 2;
        }
        oplossing = oplossing + tussenoplossing;
        i = i + 2;
    }

    return oplossing;
}

double Oefening3beste(double x) {
    return ((SinReeks(-x) + x)/pow(x, 3));
}

int main () {
    k = 0;
    while (k <= 15) {
        x = (1 / pow(10, k));
        cout << "f(x) met x=10^(-" << k << ") : " <<
        Oefening3beste(x) << endl;
        k++;
    }
}
```

En dit geeft de meest correcte oplossing.
